Elastos 内存管理对软件调试的支持

陈俞飞,陈 榕
(同济大学 基础软件工程中心,上海 200092)

摘 要:软件调试支持是操作系统的重要组成部分,而由于内存管理不善造成的软件BUG占软件故障的很大比例。从内存管理的角度,讨论了Elastos操作系统对软件调试的支持。介绍了Elastos的内存布局和堆管理器算法,然后论述了堆、栈的内存管理,以及针对软件调试所提供的栈保护页设置、堆块前/后内存越界检查等。最后简要地介绍了构件Domain技术和基于伪驱动的内核窥探调试。通过Elastos内存管理的调试支持,可以提高修复和内存管理相关的BUG,增强软件的可靠性,减小软件的开发成本。

关键词:伪驱动;Dlmalloc;Domain;堆;栈;
中图分类号:TP311 **文献标识码**:A **文章编号**:1673-629X(2010)04-0025-04

Memory Management Support for Software Debugging on Elastos

CHEN Yu-fei, CHEN Rong
(System Software Engineering Centre of Tongji University, Shanghai 200092, China)

Abstract: Software debugging support is an important part of the operation system, and software bugs, due to poor memory management, accounted for a large proportion of software fault. Argues the support of Elastos operating system for software debugging in the memory management. First, it describes the virtual memory layout and Elastos heap memory management algorithm, and then discusses the heap, stack memory management, as well as some debugging surpports, such as setting up fence page in stack, cross-border checks before/after stack chunks. Finally, it gives a brief introduction of the component Domain technology and debugging of kernel spy based on pseudo-driver. With the support of Elastos debug memory management, enhance the capacity of fixing the bug related to memory management, improve the reliability of software and reduce software development costs.

Key words: pseudo-drivers; dlmalloc; domain; heap; stack

0 引 言

内存是应用程序工作的基础,操作系统运行、应用程序加载和运行时都需要用到内存资源。Elastos操作系统是基于CAR构件的操作系统,每个Elastos应用就是一个CAR构件,当应用程序运行时,Elastos需要将其安排在内存的特定区域,以保证程序的正确运行。基于此,Elastos提供了一套内存管理机制,用来分配和管理内存资源[1]。

Elastos操作系统对虚拟内存的管理采用的是分页机制,每个页面大小为4kB。用户可以通过系统API函数_CVirtualMemory_Alloc()和_CVirtualMemory_Commit()来获取虚拟内存及映射相应的物理内存。

Elastos中,所有进程共享低2GB的内核虚拟地址空间,每个进程独享高2GB的地址空间。为了支持应用程序的运行,Elastos在虚拟内存中为每个线程设定了栈(stack)空间,为每个进程设置了堆(heap)空间。同时提供了一些策略,以支持软件调试。

1 内存管理堆算法

堆是Elastos操作系统中管理动态内存的一种机制。堆管理器(Heap Manager)从Elastos虚拟内存管理[2]机制中获取以页(page)为单位的虚拟内存,然后分割成很多个堆块来满足不同应用程序的请求。堆管理器的主要工作就是管理这些堆块,尽可能地减少堆碎片,提高堆内存的利用率。对堆块的管理常见的有两种算法:dlmalloc和二次幂算法,以下分别介绍。

1.1 dlmalloc

dlmalloc堆算法是目前应用广泛的动态内存管理

收稿日期:2009-07-23;**修回日期**:2009-10-12
基金项目:国家"863"计划资助项目(2001AA113400);国家移动通信产品研究开发专项项目(财政部(财建[2005]182号),信息产业部(信部请函[2005]297号))
作者简介:陈俞飞(1979-),男,江苏人,硕士研究生,研究方向为嵌入式操作系统、系统软件支撑技术;陈 榕,博士生导师,教授,科泰世纪首席科学家,研究方向为嵌入式系统、构件技术。

算法,它通过在堆块上增加额外的数据实现对堆块的管理。Elastos采用dlmalloc实现进程用户态内存的动态管理。

dlmalloc算法将堆划分为独立的块(chunk),并加上相应的标志位,从而便于管理。当程序要使用堆空间时,dlmalloc首先从操作系统中获得和物理页面对应的逻辑页面,然后以块的方式分配给用户。块分为使用的和空闲的两种类型,具体的内部结构参见图1,其中,P标志位表示前一内存块是否在使用,C标志位表示当前内存块是否在使用。dlmalloc可以通过P和C标志位合并非使用块和判断一个块是否在使用中。

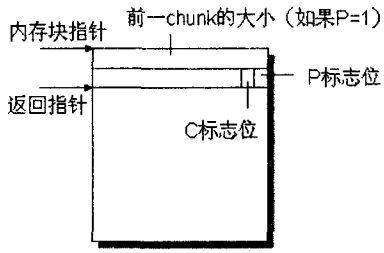

图1 块的内部结构图

dlmalloc算法主要是对空闲块的管理,而且是按块的大小分别对待的,对于小的块(小于265kB)采用的是数组加链表的方式;对于大的块,则采用树结构加链表的方式。即同样大小的块放在链表中,然后将所有的链表放在数组或树结构中。

应用程序请求内存时,dlmalloc算法按如下顺序查找空闲块:

1. 按请求的大小在数组或树结构中查找空闲块;

2. 查看Top块(堆的上边界至页边对齐处)是否可用;

3. 向系统发出虚拟内存请求,然后分配给用户;

dlmalloc还支持对mspace的管理。mspace代表的是分配给应用程序的一整块虚拟内存,然后在mspace内部,仍然采用dlmalloc算法对mspace内部进行动态分配。同时,dlmalloc还对所有的mspace进行统一管理;此时,每一个mspace相当于一个块。

在Elastos操作系统中,采用的是dlmalloc算法,dlmalloc对构件及其相应的内存请求能很好地进行管理;应用程序在加载CAR构件时,可以通过dlmalloc分配一块对应的mspace空间,再通过对应的dlmalloc接口对这块空间进行动态管理。

1.2 二次幂算法

Elastos采用二次幂算法实现操作系统内核内存的动态管理。

二次幂算法(Binary Buddy)管理的内存的最小单位是Buddy块,其大小都为2的次幂。Buddy块分割的原则是:Buddy块分割成大小相等的两块,递归下去直至恰好能满足动态请求。由于Buddy块相对起始地址的偏移都是2的次幂,所以通过很小代价的计算就可以找到一块Buddy,同时,对于总大小为2^m的Buddy块空间,只需要m个空闲链表即可;但是,这样的Buddy块内部碎片非常的高。

二次幂算法的实现步骤是:当有动态申请内存时,先将请求的大小按2的幂次方对齐,然后到空闲链表中查找对应的Buddy块以满足请求;若不存在,则从大一倍的Buddy块空闲链表取出一块,一分为二,一块给应用程序,一块插入空闲链表中。若仍不存在,则继续递归找大一倍的Buddy块空闲链表,直至失败为止。

2 栈和函数调用

Elastos操作系统中,栈是函数调用和存储局部变量所使用的连续内存区域。编译源代码的时候,编译器会将函数调用和局部变量的存取转化为合适的栈操作[3],不需要程序员编写额外的代码,故栈又称为自动内存。Elastos在创建线程时,会为每个线程创建栈,同时初始化对应的栈空间。用户线程都有两个栈,一个供线程在内核态使用,另外一个供线程在用户态使用。但是,对于同一个CPU而言只有一个栈:当前栈。

从Elastos系统的内存结构来看,用户态的栈基地址位于虚拟地址高2GB空间中,往低地址方向增长。在实际的函数调用中,调用程序首先会按特定的调用协定将实参和返回地址入栈,让后跳转到被调用函数入口处开始执行被调用函数的代码。

常用的调用协定有以下几种[4]:

* C调用协定。

C调用协定是C/C++程序的普通函数所使用的默认协定,完全使用栈来传递参数,调用函数在调用前,将参数按从左至右的顺序依次压入栈中,同时负责在函数返回后调整栈指针以清除压入栈的参数;C调用约定的关键字是_cdecl。

* 标准调用协定。

标准调用协定和C调用协定很相似,不同的是需要被调用函数清理栈中的参数。标准调用函数的关键字是_stdcall。

* 快速调用协定。

考虑到CPU访问寄存器的速度要比访问内存的速度快得多,因此,很多编译器都使用了寄存器传递参数额协定。快速调用协定的关键字是_fastcall。

* This调用协定。

Elastos系统下的程序是通过C++编译器编译的,而C++程序中的成员函数默认调用协定是This

调用协定;这种调用协定的重要的特征是 this 指针会被放入 CPU 中的寄存器内,同时也要求被调用函数清理栈,故不支持可变参数的传递。This 调用协定的关键字是_thiscall。

栈通过设置保护页面来提供软件调试的支持。Elastos 为线程初始化栈时,创建了 1MB 的栈空间,并先提交其中的 2 个页面供使用。第一个页面供线程使用,第二个页面作为栈的保护页面,为栈的自动增长和栈溢出异常的触发提供支持。当可使用的栈空间使用完时,栈顶指针会触及到栈保护页面,此时可继续提交新的页面供线程使用。若栈的 1M 空间用完,且触及栈保护页面,便触发栈溢出异常,以便调试。

3　堆操作调试

Elastos 的堆管理器根据编译时的选项来决定是否使用堆调试。如果编译选项里包含了调试选项,则提供一系列的功能来辅助调试,帮助和发现堆内存有关问题。

堆管理器提供的调试功能主要分以下几种:

* 堆尾检查(Heap Tail Checking),是在每个堆块的用户数据后增加额外的标记信息,用于检查堆块是否发生溢出。附加的数据通常为 8 个字节,也可通过全局变量来定义。如果该数据的内容遭到破坏,便说明产生了溢出,此时会产生异常,中断到调试器。

* 释放检查(Heap Free Checking),是在释放堆块时对堆进行各种检查。堆损坏很多情况下是由释放堆块引起的,比如:释放不存在的堆块、多次释放同一堆块等。释放检查能避免类似和堆块释放有关的问题。

* 参数检查[5],对传递给堆管理器的参数进行更多的检查。验证每个参数的合理性,给出调试信息。

* 调用时验证(Heap Validation on Call),即每次调用堆函数时都对整个堆进行验证和检查。开启此调试功能时,在每次调用堆函数时,堆管理器都会对头信息、段信息、堆块等进行全面的检查,及时捕获堆中的错误,达到很好的调试效果。由于此调试功能会加大堆函数的执行时间,故默认此功能是关闭的。

* 堆块标记(Heap Tagging),是为堆块增加额外的附加标记(Tag),以记录堆块的使用情况和其它信息。可以通过这些附加标记来检查堆块是否遭到破坏,若遭到破坏,便记录错误信息,进入调试状态,方便进一步的定位。

* 专门用于调试的页堆(Debug Page Heap),是为了弥补堆尾检查滞后的不足,堆管理器会在堆块后增加专门用于检测溢出的栅栏页(Fense Page),这样一旦用户数据区溢出触及栅栏页便立即会触发异常。

3.1　前/后越界调试

用户申请堆块时,堆管理器会返回堆块的地址。正常情况下,用户在所分配的堆块内操作是不会有问题的。但是,当用户使用指针不当,对堆块外的区域进行操作,这将会破坏堆上的其它数据,严重的甚至会破坏整个堆。

针对这种情况,Elastos 堆管理器增加了前/后越界调试功能,该功能通过在堆块前/后增加用于调试的栅栏页(Fence Page)(默认为 8 个字节)。当启用该功能时,一旦用户数据区溢出触及栅栏页,便会立即触发异常,进入调试状态,输出越界及对应的堆块信息。

3.2　内存泄露调试

开发 Elastos 应用程序的过程中,用户申请的内存,往往会出现未及时释放,导致内存泄漏,产生内存不足的情况。Elastos 提供了内存泄漏调试,大致采用如下的方法:

对每次成功的动态分配操作,堆管理器额外生成一个对应结点,记录堆块的相关信息(包括分配序列号 index,分配的地址 ad, 大小 size, 线程 ID 号 thd 等);所有的这些结点组成一个链表,同时由堆管理器负责维护。在释放堆块时,将对应的结点从链表中删除,以保持数据的统一。在程序退出时,若链表为空,则标识没有泄漏;反之,则说明该应用程序存在泄漏,需要进一步调试。

在实际的调试操作中,若出现内存泄漏,则控制台会输出打印信息。参考打印信息,可以设置中断条件;这样当程序加载起来后,只要满足中断条件,就会进入调试状态,方便进一步的调试跟踪。

4　构件 domain 技术

Elastos 中,每个应用程序就是一个独立的 CAR 构件[6],在运行状态时称之为域[7](domain)。Elastos 中的每个进程包含了多个域,每个域包含了独立的虚拟地址空间、独立的应用程序资源和独立的配置信息等。当构件作为域加载到进程中的时候,Elastos 系统将其安排到进程中固定的地址空间中,之所以这样做,有以下的原因:

* 由于多个域可以包含在一个进程中,所以可以保证低的系统消耗。当域之间进行切换时,由于仅仅是在进程内部的切换,带来的系统消耗就很少。

* 由于域放在进程中不同的固定的地址空间中,故可以保证域之间不会互相影响,即一个域发生错误不会影响到同一进程中的另外域或者整个进程。这有利于域之间的隔离,保证应用之间的独立性。

* 同一进程中的域,可以设置不同的安全等级。

针对不同的安全等级,可以采用不同的安全策略。

* 同一应用域中,可以包含多个线程。Elastos 应用的消息处理机制采用的就是这一机制。

* 域的错误或异常不会影响到其它的域或者整个进程。由于域是独立的,域之间的错误或异常会保持相互的独立。

* 每个域的配置信息是该域的一部分,而不属于该域所在的进程。

* 一个域中的程序终止的时候,不影响同一进程中的其它域的状态。

构件 domain 技术通过在同一进程中设置多个域,来减小应用切换带来的系统消耗,提高应用的安全性,保证应用程序能安全有效的运行。通过构件 domain 技术,可以将调试范围定位到某一个构件的内存中,方便调试。

5 基于伪驱动的内核窥探调试

一个普通的程序通常只能运行在用户态,但就调试而言,这是不够的。有时,需要获取更多的内核信息,比如:线程局部存储(TLS)的当前值,内核的一些信息。而这些信息只能通过处在内核态的线程才能取得。为了能做到这一点,需要向内核中派个亲信,称之为伪驱动。

Elastos 系统中驱动程序[8]有较高的权限,已安装使用的驱动可以获取内核的所有信息。正因这样,可以编写一个伪驱动,让其运行于内核态,获取无法从用户态获取的信息,帮助更好的调试。调试流程如下:

* 运行应用程序、调试程序和伪驱动;

* 当应用程序需要获取内核态的信息时,则通过调试程序调用伪驱动的相关接口,以获取相应的内核信息,用户可以根据这些信息,观察内核中的信息是否异常,以采取进一步的调试策略。

6 结束语

在 Elastos 内存管理体系结构中,利用栈、堆、构件 domain 和伪驱动的内核窥探技术,对软件的调试提供良好的支持。通过这些支持,用户可以在开发 Elastos 程序中,方便地定位程序在内存方面的错误。同时,用户亦可以在此基础上,结合 Elastos 平台的其它特性,增强对程序的调试能力,提高软件的可靠性。

参考文献:

[1] Koretide. CAR's Manual[EB/OL]. 2009 - 07. http://www.koretide.com.cn.

[2] 王 铮,李志军.一种适用嵌入式系统的自适应动态内存管理方案[J].计算机技术与发展,2007,17(3):48 - 54.

[3] 王明路,王希敏,王 哲.嵌入式系统中池式内存分配方法的分析[J].计算机与数字工程,2008(2):57 - 61.

[4] Fog A. Calling conventions for different C++ compilers and operating systems[M].[s.l.]:[s.n.],2009:15 - 23.

[5] 袁 宁.基于共享主存计算机的含错与动态检查点技术研究[D].长沙:国防科学技术大学,2006:42 - 50.

[6] Chen Rong. The Application of Middleware Technology in Embedded OS[C]//Workshop on Embedded System, in Conjunction with the ICYCS(6th). Hangzhou:[s.n.],2001.

[7] 白轶荣,陈 榕. Elastos 平台上可执行文件的三种入口规范[J].计算机技术与发展,2008,18(11):220 - 222.

[8] 杨向科,陈 榕.基于 Elastos 的构件化驱动编程模型的研究[J].计算机技术与发展,2008,18(12):25 - 31.

(上接第 24 页)

的参数规律,进一步地优化算法,并且给出算法的仿真结果。

参考文献:

[1] IETF. Mobile Ad hoc Network Chapter[EB/OL]. 2003. http://www.ietf.org.

[2] 董传杰,王汝传. Ad hoc 网络路由问题的研究[J].南京邮电学院学报,2005,25(3):67 - 72.

[3] Charles E, Perkins, Elizabeth M, et al. Ad hoc on Demand Distance Vector Routing[C]// Proceedings WMCSA'99. Second IEEE Workshop on Mobile Computing Systems and Applications. Los Alamitos, CA, USA:[s.n.], 1999:90 - 100.

[4] Ehsan H, Uzmi Z A. Performance Comparison of Ad hoc Wireless Network Routing Protocols[J]. INMIC IEEE, 2004,9(4):450 - 465.

[5] Corson S, Macker J. Mobile Ad hoc networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations[S]. RFC 2501, 1999.

[6] Caro G D, Dorigo M. An adaptive multi - agent routing algorithm inspired by ants behavior[C]//In: Fifth Annual Australasian Conference on Parallel and Real - Time Systems. Adelaide, Australia:[s.n.],1998:28 - 29.

[7] Foundation for Intelligent Physical Agents (FIPA)[EB/OL]. 2005. http://www.fipa.org.

[8] 王汝传,徐小龙,黄海平.智能 Agent 及其在信息网络中的应用[M].北京:北京邮电大学出版社,2006.

[9] Wooldridge M. An Introduction to Multi - agent Systems [M]. Chichester, England: John Wiley & Sons, 2002:10 - 31.